DLP Lab3 Report

Shu Kai, Lin

March 2025

1 Introduction

In this lab, I implemented a VQGAN-based image tokenizer and trained a transformer model to predict masked visual tokens. After training the transformer, I applied it to perform iterative inpainting using learned latent representations. I experimented with different masking strategies and scheduling methods to improve the reconstruction quality. Finally, I achieved the best FID score of 38.46 using cosine-based mask scheduling.

2 Implementation Details

2.1 MultiHeadAttention

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.dim = dim
        self.key = nn.Linear(dim, dim)
        self.query = nn.Linear(dim, dim)
        self.value = nn.Linear(dim, dim)
        self.attn_drop = nn.Dropout(attn_drop)
        self.output = nn.Linear(dim, dim)
    def forward(self, x):
        B, T, D = x.shape
        head_dim = D // self.num_heads
        q = self.query(x).view(B, T, self.num_heads, head_dim).transpose(1, 2)  # (B, H, T, D_head)
        k = self.key(x).view(B, T, self.num_heads, head_dim).transpose(1, 2)
        v = self.value(x).view(B, T, self.num_heads, head_dim).transpose(1, 2)
        attn_scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(head_dim,
        ↪  dtype=q.dtype, device=q.device))  # (B, H, T, T)
        attn_weights = torch.softmax(attn_scores, dim=-1)
        attn_weights = self.attn_drop(attn_weights)
        context = torch.matmul(attn_weights, v)  # (B, H, T, D_head)
        context = context.transpose(1, 2).contiguous().view(B, T, D)
        out = self.output(context)
        return out
```

The MultiHeadAttention class implements multi-head self-attention , which processes an input tensor x of shape (B, T, D), where B is the batch size, T is the sequence length, and D is the embedding dimension. First, x is passed through three linear layers to produce queries, keys, and values, each still of shape (B, T, D). These are then reshaped and transposed into the shape (B, num_heads, T, head_dim) so that each of the multiple heads can process a portion of the input features independently. Attention scores are computed using the scaled dot product of the queries and keys, resulting in a tensor of shape (B, num_heads, T, T), which is then normalized using softmax to obtain attention weights. Dropout is applied to these weights for regularization. The weighted sum of the values is then calculated using these weights, producing context vectors that represent attended information for each head. These context vectors are transposed and reshaped back to the original input shape (B, T, D) by concatenating the heads, and then passed through a final linear projection layer. As a result, the input x is transformed into a new representation where each token can gather contextual information from every other token in the sequence.

2.2 Training Transformer

2.2.1 train_one_epoch

```
    def train_one_epoch(self, train_loader, epoch):
        self.model.train()  # Set model to training mode
        losses = 0.0
        scaler = torch.cuda.amp.GradScaler()
        progress_bar = tqdm(train_loader, desc=f"Epoch {epoch}/{self.tol_epochs}", leave=True)
        for img in progress_bar:
            img = img.to(self.device)
            self.optim.zero_grad()
            with torch.cuda.amp.autocast():
                pred_y, y = self.model(img)
                loss = F.cross_entropy(pred_y, y)
            scaler.scale(loss).backward()
            scaler.step(self.optim)
            scaler.update()
            losses += loss.item()
            lr = self.optim.param_groups[0]["lr"]
            progress_bar.set_postfix(loss=f"{loss.item():.4f}")
        self.scheduler.step()
        return losses / len(train_loader)
```

Using FP16 scalar to speedup training and uses logits and ground truth logits calculate cross entropy as loss.

2.2.2 eval_one_epoch

```
    def eval_one_epoch(self, val_loader):
        self.model.eval()
        losses = 0.0
        with torch.no_grad():
            for i, img in tqdm(enumerate(val_loader), total=len(val_loader)):
                img = img.to(self.device)
                pred_y, y = self.model(img)
                loss = F.cross_entropy(pred_y, y)
```

```
                losses += loss.item()
        return losses / len(val_loader)

```

2.2.3 Configure scheduling

```
    def configure_optimizers(self, epoches, lr,  warmup_steps, eta_min=0.0):
        torch.optim.AdamW(self.model.parameters(),lr=lr)
        optimizer = torch.optim.AdamW(self.model.parameters(), lr=lr, betas=(0.9, 0.95))
        def lr_lambda(current_step):
            if current_step < warmup_steps:
                return float(current_step) / float(max(1, warmup_steps))
            else:
                progress = float(current_step - warmup_steps) / float(max(1, epoches - warmup_steps))
                return eta_min + (1 - eta_min) * 0.5 * (1 + np.cos(np.pi * progress))

        scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda)
        return optimizer,scheduler
```

The configure_optimizers function sets up the AdamW optimizer and a learning rate scheduler for training a model. It first defines the optimizer using the model's parameters, a specified learning rate, and custom beta values (0.9, 0.95). Then, it defines a lambda function to control how the learning rate changes during training. In the initial warmup_steps, the learning rate increases linearly from 0 to the base learning rate, helping stabilize early training. After warm-up, it follows a cosine annealing schedule where the learning rate smoothly decays toward a minimum factor (eta_min * lr) over the remaining training steps.

2.2.4 Forward

```
    def forward(self, x):
        _, z_indices=self.encode_to_z(x)
        z_indices = z_indices.view(-1, self.num_image_tokens) # shape: (B, T)
        B,T = z_indices.shape
        mask_ratio = np.random.uniform(0, 1)
        if self.prob_type == "topk":
            num_keep = math.floor(mask_ratio * T)
            rand_scores = torch.rand(B, T, device=z_indices.device)
            topk_indices = rand_scores.topk(num_keep, dim=1).indices  # [B, num_keep]
            mask = torch.zeros(B, T, dtype=torch.bool, device=z_indices.device)
            mask.scatter_(1, topk_indices, True)  # True = keep, False = mask
        else:
            mask = torch.bernoulli(torch.full_like(z_indices, 1 - mask_ratio, dtype=torch.float)).bool()
        masked_input = z_indices.clone()
        masked_input[~mask] = self.mask_token_id
        logits = self.transformer(masked_input)
        logits = logits[..., :self.mask_token_id]
        ground_truth = torch.zeros(z_indices.shape[0],                                z_indices.shape[1],
                        .to(z_indices.device)
                        .scatter_(2, z_indices.unsqueeze(-1), 1)
        return logits, ground_truth
```

Forward method carries out masked visual token modeling within a VQGAN Transformer architecture by converting input images into discrete latent tokens and training the transformer to predict

masked tokens. The process begins by encoding the input image x using a VQGAN encoder via self.encode_to_z(x), which returns discrete token indices z_indices representing the image content. These indices are reshaped to shape (B, T), where B is the batch size and T is the number of tokens per image (flattened spatial layout). A random mask_ratio between 0 and 1 is sampled, indicating the proportion of tokens to mask. If self.prob_type is "topk", the method determines a fixed number of tokens to keep (based on the mask_ratio), assigns random scores to all tokens, and keeps only those with the top scores, generating a binary mask indicating which tokens remain visible. If "topk" is not selected, a Bernoulli distribution is used to probabilistically decide for each token whether it should be masked or kept, using the same ratio. A masked version of the input sequence is created by replacing all masked tokens in z_indices with a special self.mask_token_id. This masked sequence is then passed through the transformer to produce logits of shape (B, T, vocab_size), which are optionally truncated to exclude the mask token from being predicted. For supervision, the ground truth token indices are converted to one-hot vectors using scatter_, resulting in a tensor of shape (B, T, vocab_size) that aligns with the predicted logits. The method returns both the logits and the ground_truth for use in computing the loss during training, encouraging the transformer to learn to accurately recover masked visual tokens and thus understand the structure and semantics of images in latent space.

2.3 Decoder

2.3.1 Inpainting

```python
    @torch.no_grad()
    def inpainting(self,image,ratio,mask_b):
        _, z_indices = self.encode_to_z(image)
        mask_token_tensor = torch.full_like(z_indices, self.mask_token_id)
        z_indices_input = torch.where(mask_b == 1, mask_token_tensor, z_indices)
        logits = self.transformer(z_indices_input)
        probabilities = torch.nn.functional.softmax(logits, dim=-1)
        z_indices_predict_prob, z_indices_predict = torch.max(probabilities, dim=-1)
        ratio = self.gamma(ratio)
        g = -torch.log(-torch.log(torch.rand_like(z_indices_predict_prob)))
        temperature = self.choice_temperature * (1 - ratio)
        confidence = z_indices_predict_prob + temperature * g
        confidence = torch.where(mask_b == 0,
                                 torch.tensor(float('inf'), device=mask_b.device),
                                 confidence)
        n = math.ceil(mask_b.sum().item()* ratio)
        flat_confidence = confidence.view(-1)
        _, idx_to_mask = torch.topk(flat_confidence, n, largest=False)
        flat_mask_bc = torch.zeros_like(flat_confidence, dtype=mask_b.dtype)
        flat_mask_bc[idx_to_mask] = 1
        mask_bc = flat_mask_bc.view_as(mask_b)
        mask_bc = mask_bc * mask_b
        return z_indices_predict, mask_bc
```

Inpainting method performs a single step of iterative visual token inpainting using a masked image representation in the latent space of a VQGAN Transformer. It begins by encoding the input image into discrete token indices z_indices using the VQGAN encoder. Createing a tensor filled with the special mask_token_id, then uses the provided binary mask mask_b to replace masked positions in z_indices with the mask token, producing z_indices_input. This masked input is fed into the

transformer to generate logits, which are converted to probabilities using softmax. The model selects the most probable token at each position using torch.max, giving both the predicted indices and their confidence scores. A gamma-decayed ratio is computed (via self.gamma(ratio)) to control how many of the masked tokens will be updated in this step. To add sampling noise, Gumbel noise is applied to the confidence scores using the formula -log(-log(U)), where U is uniform noise; this makes token selection more stochastic. A temperature-scaled version of this noise is added to the confidence to balance between certainty and randomness. Positions that were not originally masked are given infinite confidence to ensure they are never selected for updating. The method then determines how many tokens to update (n), selects the n lowest-confidence masked positions using topk, and constructs a new update mask mask_bc that identifies which masked tokens should be regenerated in this step. It returns the transformer's predictions z_indices_predict and the new partial mask mask_bc, which will be used in iterative inpainting to progressively refine the image.

2.3.2 Gamma Function

```
    def gamma_func(self, mode="cosine"):
        def linear(ratio):
            return 1.0 - ratio
        def cos(ratio):
            return np.cos(ratio * np.pi / 2)
        def square(ratio):
            return 1 -  np.square(ratio)
        if mode == "linear":
            return linear
        elif mode == "cosine":
            return cos
        elif mode == "square":
            return square
        else:
            raise NotImplementedError
```

In VQGAN Transformer decoding, the model may gradually mix between reconstructed (ground truth) tokens and predicted tokens as decoding progresses. The gamma function returns a value between 1 and 0 that determines how much weight to give to the original tokens versus the model's predictions at a given step. This will only be use and inpainting session while training session will use uniform distributed probability.

2.3.3 Inference

```
    for step in range(self.total_iter):
        if step == self.sweet_spot:
            break
        ratio = (step+1)/self.total_iter #this should be updated
        z_indices_predict, mask_bc = self.model.inpainting(image, ratio, mask_bc)
        mask_i=mask_bc.view(1, 16, 16)
        mask_image = torch.ones(3, 16, 16)
        indices = torch.nonzero(mask_i, as_tuple=False)#label mask true
        mask_image[:, indices[:, 1], indices[:, 2]] = 0
        maska[step]=mask_image
        shape=(1,16,16,256)
        z_q = self.model.vqgan.codebook.embedding(z_indices_predict).view(shape)
```

```
        z_q = z_q.permute(0, 3, 1, 2)
        decoded_img=self.model.vqgan.decode(z_q)
        dec_img_ori=(decoded_img[0]*std)+mean
        image = decoded_img
        imga[step+1]=dec_img_ori
```

In inferencing, it does iterative decoding for total_iter times and return the final predicted laten tokens. Also it will be break at sweet_spot time.

3 Discussion

3.1 Iterations of Mask Scheduling

In VQGAN Transformer-based inpainting, using an overly long mask scheduling—where only a small portion of tokens are updated per step—can lead to worse outcomes because the model becomes too dependent on its own early, uncertain predictions, allowing errors to accumulate across iterations. This slow update prevents the model from making global adjustments efficiently, resulting in weaker overall structure and coherence in the generated image. Additionally, since each step may involve Gumbel noise for sampling, long schedules provide more opportunities for noise to distort predictions without sufficient correction. The model also underutilizes its capacity, making minimal progress per step. As a result, longer schedules often lead to blurrier, less realistic outputs, and empirically yield worse FID scores compared to moderately paced schedules that balance refinement with stability. The prove is shown below where in experiment with 21 tol and sweet iteration fid is above 45 and in best with 1 tol and sweet iteration fid is below 38.

3.2 Transformer Performance

During the training session of transformer, I discovered that 100 epoch is the best. With too much or too less epoch will cause the transformer overfit or underfit .

Figure 1: Result of Training 20 epochs, FID = 49.17

Figure 2: Result of Training 50 epochs, FID = 43.62

Figure 3: Result of Training 100 epochs, FID = 39.96

4 Experiment Score

4.1 Show iterative decoding

4.1.1 Linear

FID: 47.6045

Figure 4: Linear Mask in Latent domain

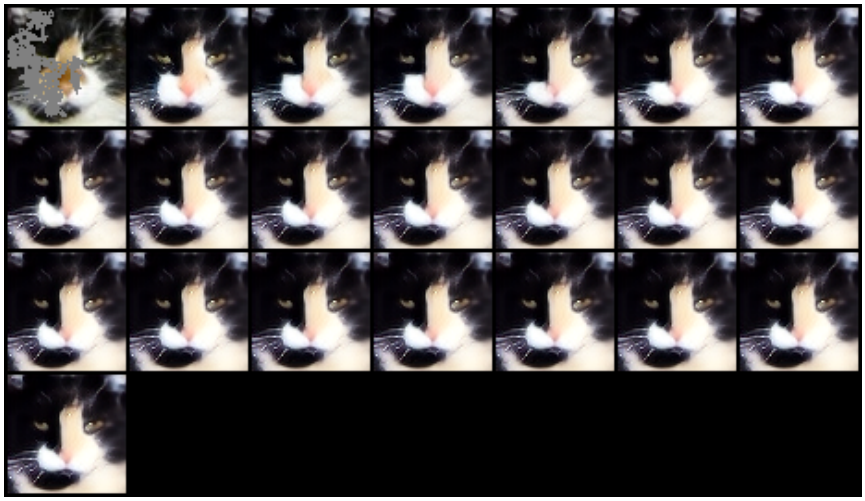


Figure 5: Linear Predicted images

4.1.2 Square

FID: 48.5210

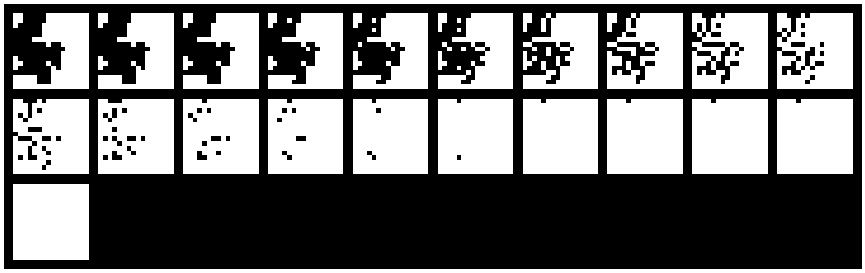


Figure 6: Square Mask in Latent domain

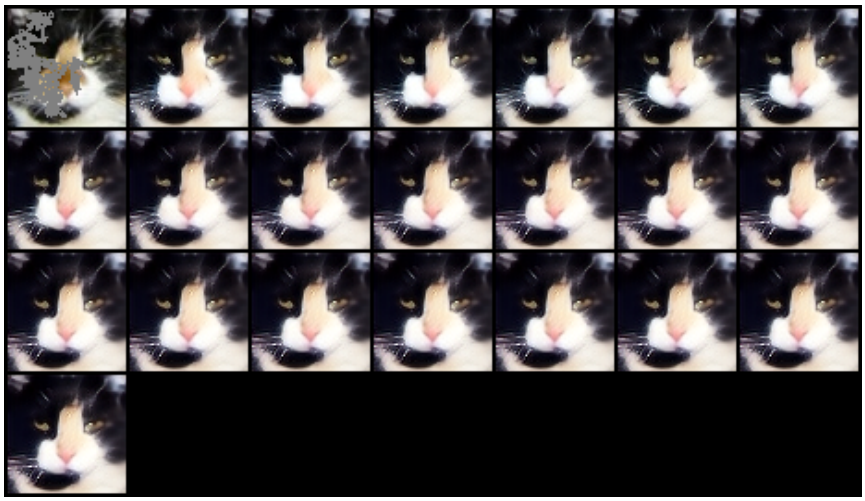


Figure 7: Square Predicted images

4.1.3 Cosine

FID: 48.69170286745373

Figure 8: Cosine Mask in Latent domain

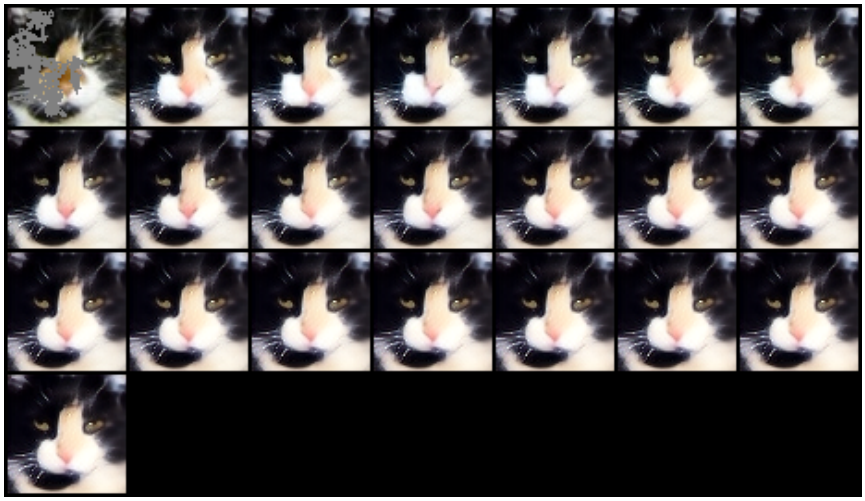


Figure 9: Consine Predicted images

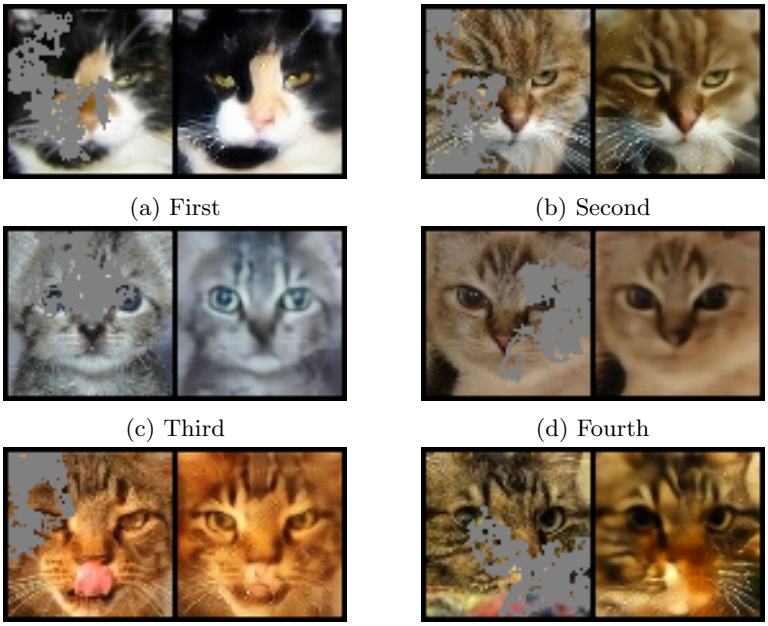
4.2 Best FID

4.2.1 Screenshot

```
747
100% 15/15 [00:03<00:00,  3.85it/s]
100% 15/15 [00:03<00:00,  4.78it/s]
FID:  35.7647923772887
```

Figure 10: Best FID

4.2.2 Masked Images v.s MaskGIT Inpainting Results



(a) First (b) Second

(c) Third (d) Fourth

(e) Fifth (f) Sixth

Figure 11: Masked Images v.s MaskGIT Inpainting Results

4.2.3 Hyperparameter

Training Transformer:

- learning-rate: 5e-5
- Batch-size: 150
- num_workers: 32
- epochs: 100
- eta: 1e-6
- warmup_step: 10
- random seed: 42

Inpainting Parameters:

- sweet_spot: 1
- total_iter: 5
- choice_temparture: 4.5

- mask_func: cosine
- random seed: 42